PATENT ABSTRACTS OF JAPAN

(11)Publication number : 05-265770
(43)Date of publication of application : 15.10.1993

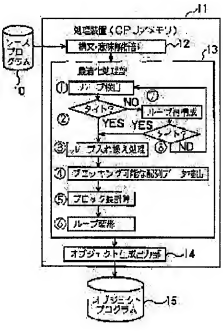
(51)Int.Cl. G06F 9/45

(21)Application number : 04-061984
(22)Date of filing : 18.03.1992

(71)Applicant : FUJITSU LTD
(72)Inventor : NAKAHIRA NAOJI

(54) COMPUTER LANGUAGE PROCESSING METHOD

(57)Abstract:
PURPOSE: To improve the execution performance of a program by performing optimization, by which data in a multiplex loop can be accessed at a high speed, in a wide range.
CONSTITUTION: With respect to the computer language processing method which performs the optmization processing at the time of translation to an object program 15 of a program 10 which is operated on a computer having a hierarchical memory, loops having a tight structure are detected in an optimization processing part 13 to recognize loops which can be blocked. Data overlap analysis is used to substitute loops so that the access distance of array data in the innermost loop is shortened. Candidate arrays which can be blocked are extracted, and a division object array and a division object index are determined from candidate arrays. The divided block length is so determined that arrays after division can be stored in a cache memory, and the loop is transformed.



(19)日本国特許庁（ＪＰ）　(12)**公開特許公報（Ａ）**　(11)特許出願公開番号

特開平5－265770

(43)公開日　平成５年(1993)10月15日

(51)Int.Cl.[5]	識別記号	庁内整理番号	ＦＩ	技術表示箇所
Ｇ０６Ｆ　9/45				
		9292－5Ｂ	Ｇ０６Ｆ　9/44　３２２　Ｇ	

審査請求　未請求　請求項の数２(全 10 頁)

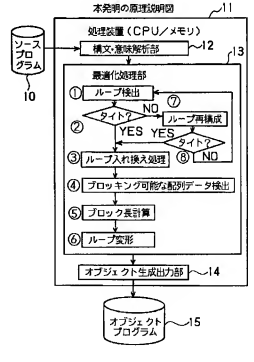
(21)出願番号　特願平4－61984

(22)出願日　平成４年(1992)３月18日

(71)出願人　000005223
富士通株式会社
神奈川県川崎市中原区上小田中1015番地
(72)発明者　中平　直司
神奈川県川崎市中原区上小田中1015番地
富士通株式会社内
(74)代理人　弁理士　小笠原　吉義　（外２名）

(54)【発明の名称】　計算機言語処理方法

(57)【要約】
【目的】本発明は，階層型メモリを有する計算機上で動作するプログラム10について，オブジェクトプログラム15への翻訳時に最適化処理を行う計算機言語処理方法に関し，多重ループ中のデータを高速にアクセスすることができる最適化を広範囲に実施し，プログラムの実行性能を向上させることを目的とする。
【構成】最適化処理部13において，タイトな構造を持つループを検出し，ブロッキング可能なループを認識する。そして，データの重なり解析を利用することにより，最内ループにおける配列データのアクセス距離が短くなるようにループを入れ換える。次に，ブロッキング可能な候補配列を抽出し，候補配列の中から分割対象配列と分割対象インデックスを決定する。分割後の配列がキャッシュメモリのサイズに収まるように，分割のブロック長を決定し，ループの変形を行う。



1

【特許請求の範囲】

【請求項１】　多重ループを含むソースプログラム(10)を，階層型メモリを有する計算機上で動作するオブジェクトプログラム(15)に変換する計算機言語処理方法において，タイトな構造を持つループを検出し，ブロッキング可能なループを認識する処理過程（①，②）と，データの重なり解析を利用することにより，インデックス交換が可能なループの範囲を識別し，最内ループにおける配列データのアクセス距離が短くなるように外側ループと最内ループとを入れ換えて，インデックス交換を実施する処理過程（③）と，ループの深さとインデックスとの関係，最内ループの配列をアクセスするインデックスとループとの関係により，ブロッキング可能な候補配列を抽出し，候補配列の中から分割対象配列と分割対象インデックスを決定する処理過程（④）と，分割後の配列がキャッシュメモリのサイズに収まるように，キャッシュメモリのサイズおよびループ回転数に基づいて，分割のブロック長を決定する処理過程（⑤）と，分割すべき配列データ，分割すべきインデックスおよび分割のブロック長をもとに，分割数を指示するループを元のループの外に生成し，ループの変形を行う処理過程（⑥）とを有し，生成するオブジェクトプログラムの最適化処理を行うことを特徴とする計算機言語処理方法。

【請求項２】　請求項１記載の計算機言語処理方法において，ブロッキング対象ループがタイトなループ構造でない場合に，ループ分割によってタイトなループ構造に再構成する処理過程（⑦）を有することを特徴とする計算機言語処理方法。

【発明の詳細な説明】

【０００１】

【産業上の利用分野】本発明は，階層型メモリを有する計算機上で動作するプログラムについて，オブジェクトプログラムへの翻訳時に最適化処理を行う計算機言語処理方法に関する。

【０００２】最近のハードウェアは，主記憶の他に，より高速なメモリ（以下，キャシュメモリという）を実装した階層型メモリ構造を持つものが一般的である。アクセス頻度の高いデータは，キャシュメモリに格納され高速にアクセスされる。一般に，キャシュメモリと主記憶のアクセスとの差は数～数十マシンサイクルと言われており，主記憶の方が圧倒的にメモリアクセスのコストが大きい。

【０００３】従って，高級言語で記述されたプログラムをコンパイルする際に，できるだけキャッシュメモリが有効利用されるように最適化を行えば，プログラムの実行性能が向上する。そのための適切な技術手段が必要とされる。

【０００４】

【従来の技術】従来の最適化コンパイラでは，ユーザが記述したソースプログラムに対して，命令数の削減／命令の並べ換えにより，メモリアクセスの縮小化／パイプラインにおけるディレイの縮小化を実施し，処理の向上を図ってきた。

2

【０００５】しかし，頻繁にメモリアクセスが必要な配列データ等のデータをキャシュメモリに保持する方法は，ほとんどの場合，ハードウェアのキャシュメモリの管理方式に処理が委ねられている。ハードウェアで実現されているキャシュメモリの制御方式では，ソースプログラム上でキャシュメモリのミスヒットが認識できる場合でも，キャシュメモリへのデータの配置・更新が一意的に処理される。このため，ソフトウェアにより，ユーザプログラムの特徴に合わせてデータをキャシュメモリ上へ配置することが，性能向上の観点から重要な問題となる。

【０００６】上記の問題点をソフトウェアで解決する手段として，ブロッキングと呼ばれる最適化技術が存在する。ブロッキング最適化の目的は，多重のループで構成されるループ構造に対して，その最内ループで使用される配列をキャシュサイズに収まるように分割して，配列データが常にキャシュメモリ上に保持されるようにループの変形を行うことにある。

【０００７】図１０の（イ）にブロッキングの最適化が適用可能なループの例を示す。このループでは，配列ｂ（ｋ，ｊ）は，インデックスｉが更新される毎に，同一の配列要素をアクセスする。従って，最内ループにおける配列ｂに対するメモリアクセスの軌跡は，図１０の（ロ）に示すように，同じ軌跡が繰り返される。

【０００８】もし，最内ループの配列ｂのサイズがキャシュサイズを超えているならば，一定間隔ごとに，キャシュメモリ上でミスヒットが生じることになる。キャシュメモリを有効に利用するためには，配列ｂをキャシュサイズに格納できる大きさに分割し，分割を指示するループを，最外ループの外に生成することが必要となる。

【０００９】現状のブロッキングによる最適化では，以下の問題点があった。

(1) 通常のユーザプログラムでは，ループ変形を実施せずに，ブロッキングの最適化を実施できることは非常に稀である。そのため，本最適化機能をより広範囲に適用するためには，インデックス交換／ループ分割といった，各種のループ変形の最適化との融合が必要になる。この適用順序によって，ブロッキングが適用できるループが認識できる場合とできない場合がある。しかしながら，従来技術では，これらを考慮する処理が行われていないので，広範囲に最適化を適用することはできなかった。

【００１０】(2) ブロッキング可能なデータが検出できた場合，どのインデックスで配列を分割するかの決定が性能向上に大きな影響を及ぼす。どのインデックスで配列データを分割するかは，配列データの大きさ，対象候補の個数およびループの実行順序関係に依存するが，従

来の処理系では，発見的な方法を取っているため，適切な分割がなされないことがあった。

【0011】(3) ハードウェアに実装されているキャシュサイズとブロッキング対象となった配列データとから，最適な分割値（またはブロック長）を決定する必要がある。しかしながら，複数の配列データが分割対象となった場合を含めて最適な分割値の決定方式が確立されておらず，適切な分割値が得られないことがあった。

【0012】

【発明が解決しようとする課題】本発明は上記問題点の解決を図り，翻訳対象プログラムについて，多重ループ中のデータを高速にアクセスすることができる最適化を広範囲に実施し，かつ適切な分割を可能とすることにより，プログラムの実行性能を向上させることを目的とする。

【0013】

【課題を解決するための手段】図1は，本発明の原理説明図である。図1において，10はコンパイル対象となるソースプログラム，11はコンパイラが動作するCPUおよびメモリなどからなる処理装置，12はソースプログラム10を入力し，構文解析および意味解析を行う構文・意味解析部，13は構文・意味解析部12の処理結果による中間テキストについて最適化を行う最適化処理部，14は命令スケジューリングやレジスタ割り付けを行い，オブジェクトコードを生成して出力するオブジェクト生成出力部，15はコンパイル結果のオブジェクトプログラムを表す。

【0014】本発明では，主記憶とキャッシュメモリの階層型メモリを有する計算機をターゲットとするソースプログラム10をコンパイルする際に，最適化処理部13において以下の処理を行う。

【0015】① タイトな構造を持つループを検出し，ブロッキング可能なループを認識する。タイトな構造とは，ループ制御変数以外のループ回帰変数がないこと，ループ外への飛び出しがないこと，手続き呼出しが存在しないことである。

【0016】② 検出したループがタイトな構造を持たない場合，処理⑦を行う。

③ データの重なり解析を利用することにより，インデックス交換が可能なループの範囲を識別し，最内ループにおける配列データのアクセス距離が短くなるように外側ループと最内ループとを入れ換えて，インデックス交換を実施する。

【0017】④ ループの深さとインデックスとの関係，最内ループの配列をアクセスするインデックスとループとの関係により，ブロッキング可能な候補配列を抽出し，候補配列の中から分割対象配列と分割対象インデックスを決定する。

【0018】⑤ 分割後の配列がキャッシュメモリのサイズに収まるように，キャッシュメモリのサイズおよびループ回転数に基づいて，分割のブロック長を決定する。

⑥ 分割すべき配列データ，分割すべきインデックスおよび分割のブロック長をもとに，分割数を指示するループを元のループの外に生成し，ループの変形を行う。

【0019】⑦ 検出したループがタイトな構造でない場合には，ループ分割によってタイトなループ構造に再構成する。

⑧ タイトなループ構造への再構成が成功したならば，処理③以下を実行する。失敗したならば，そのループの最適化を諦め，次のループの検出を行う。

【0020】処理⑥によるループ変形によって，キャッシュメモリにおけるミスヒットを減少させ，実行性能を向上させることができる。

【0021】

【作用】本発明は，以下の方式を採用することにより，従来技術によるブロッキングの問題点を改善するものである。

【0022】(1) ブロッキング可能な配列を認識する場合に，ループ分割，インデックス交換等のループ変形の最適化を利用して，ブロッキングの適用範囲を広げる。(2) 最適な分割値（またはブロック長）を決定する。

【0023】ブロッキング可能な配列データを分割する場合には，ループの回転数で割り切れて，かつキャシュメモリに十分に収まる大きさとなるように分割値を決定する。また，ブロッキング可能な配列データが複数存在する場合においても，複数の配列データがキャシュサイズに十分収まるように，最適な分割値を決定する。

【0024】これによって，主記憶へのアクセスを少なくした効率のよいオブジェクトプログラムを生成することが可能になる。

【0025】

【実施例】以下，図1に示す最適化処理部13における本発明に関係する部分の実施例の処理を，詳細に説明する。

【0026】1) ブロッキング可能なループの認識

ブロッキングにより，実行結果が変わらない条件は，最内ループに対して，(1) ループ制御変数以外のループ回帰変数がない，(2) ループ外への飛び出しがない，(3) 手続き呼出しが存在しない，という3つの要件を満たすタイトなループ構造を持つことである。また，この条件に付け加えて，ブロッキングの最適化処理が実施可能であるためには，多重ループにおいて，最内ループの配列の演算が最外ループのインデックスに依存せず，最外ループのインデックスが変化しても，最内ループでは，常にデータアクセスの法則性が変化しないことが必要である。

【0027】ブロッキングの対象ループは，上述したように，タイトなループに対してのみ実施されるため，対象となる多重ループがタイトなループでない場合，タイ

トなループ構造への再構成を実施する。この処理により，ブロッキング可能なループの適用範囲を広げることができる。

【0028】図2は，そのループ分割によるループ再構成の例を示している。ループ分割とは，図2の（イ）に示すように，1つのループをそれぞれ同じループ制御変数を持つ複数のループに分ける処理である。本発明では，タイトでない多重ループを複数のタイトなループに分割することにより，タイトなループの最適化を可能とする。タイトな多重ループとは，ループが直列の入れ子をなし，最内のループのみに，実行文が出現する多重ループのことを言う。

【0029】ループ分割は，以下の手順で行う。

(1) タイトなループの検出

最内ループから最外ループへとループを検出し，最内ループにしか実行文が存在しないループをタイトなループとして認識する。それ以外のループは，タイトなループとして検出できないので，タイトでない多重ループを複数のタイトなループに分割する必要がある。

【0030】(2) タイトでない多重ループの検索と分割対象ループの検索

最内ループから外側ループへと検索し，以下のいずれかの条件を満足するループをループ分割の対象とする。

【0031】－最内ループと外側ループとの間に実行文（ループ制御のための命令は除く）が存在する。

－同じ深さのタイトな兄弟ループが存在する。

【0032】－ループ内からの飛び出しが存在しない。

(3) ループ分割

例えば，図2（ロ）の(a) に示す多重ループは，最内ループと外側ループとの間に実行文Aが存在する。❸のループから検索すると，❷と❸のループの間に，実行文Aが存在することがわかるので，❷がループ分割の対象となる。このループ分割により，ループは図2（ロ）の(b) に示すようになる。

【0033】図2（ハ）の(a) に示す多重ループは，同じ深さのタイトな兄弟ループが存在する例である。❶のループから検索する。❶，❷のループは分割の対象とはならない。双方を検索した後，❶と❷を比較すると，同じ深さのタイトな兄弟ループであることがわかるので，図2（ハ）の(b) に示すようにループ分割を行う。

【0034】2）ループ入れ換え処理

2.1) インデックス交換可能ループの判定

例えば，多次元配列がタイトな多重ループ内で，最内ループの制御変数が一次元目以外の次元を引用している場合，一次元目の配列要素を引用している外側ループが最内ループになるようにループを入れ換える処理を，ループのインデックス交換という。インデックス交換を実施することにより，メモリ上の連続領域をアクセスすることになり，メモリアクセス効率が向上する。

【0035】例えば図3の（イ）に示すループでは，（ロ）に示す①，②，③，…の順番で配列Aの各要素にアクセスすることになる。従って，不連続領域へのアクセスとなる。配列Bも同様である。なお，この飛び飛びにアクセスするデータの間隔を，配列データのアクセス距離という。

【0036】インデックス交換により，図3の（ハ）に示すように，最内ループと外側ループとを入れ換えると，アクセスの順番は，（ニ）に示す①，②，③，…のようになり，最内ループの配列A（I，J），B（I，J）は，連続領域のアクセスとなる。

【0037】インデックス交換の目的は，このように配列データのアクセス距離が最小となるようにすることである。インデックス交換が可能であるかどうかは，ループ中の配列要素の依存関係による。そこで，“データの重なり解析”により，インデックス交換が可能なループの範囲を識別し，インデックス交換を実施する。

【0038】データの重なり解析とは，次のような処理である。一般に，多重ループのインデックス交換やループ分割等のループの構造を大幅に変換する最適化では，そのループ中の配列要素の依存関係を認識する必要がある。このとき，ループ内の配列要素の振る舞いを，添字と制御変数の関係から単純化した形にすることで，データの重なり（依存）関係を調べて，上記で述べた最適化処理が実施できるかどうかを判断する。データの重なり解析は，このようなときに行う処理であり，具体的には次の情報を解析する処理である。

【0039】(1) 配列要素のインデックス空間上での定義・参照の順序関係。

この情報解析では，個々の配列要素の定義・参照情報の他に，最内ループで演算を実施する異なる配列要素同士が，互いに同一領域をアクセスするかの情報も収集する。

【0040】(2) 単純変数同士の演算における先行順序関係。

以上のデータの重なり解析処理は，ベクトル化コンパイラで採用されているデータの依存関係の解析と同様でよく，よく知られている処理であるので，ここでの説明はこの程度にとどめる。

【0041】もし，インデックス交換が不可能なループであれば，現状のインデックスの並びで，ブロッキング可能かどうかの判定を行う。本方式を以下のような順序に拡張することにより，ブロッキングを実施する配列が決定した後に，インデックス交換を実施することが可能である。

【0042】(1) インデックス交換をする部分を，交換できるインデックス空間に置き換える。

(2) “ブロッキング可能な配列データ”の候補集合を求める。

【0043】(3) “交換できるインデックス空間”でインデックス交換を実施する。

(1) ～(3) の操作を繰り返し実施することにより，よりきめこまかなブロッキングの処理を実現できる。

【００４４】３）ブロッキング可能な配列データの検出

3.1) 配列データ再利用の解析

ループの深さとインデックスとの関係，最内ループの配列をアクセスするインデックスとループとの関係により，ブロッキング可能な配列を決定する。

【００４５】本解析方法は，“データの重なり解析”のように，最内ループで主記憶アクセスされる定義と参照のデータが重なっているか等の情報は一切収集しない。ブロッキングに必要なデータの依存関係の解析は，解析中のインデックスXに対して，最内ループで，Xの値が変化した時，同一順序の配列のアクセスが存在するかどうかだけを調べればよい。

【００４６】この配列データ再利用の解析処理についての詳しい説明は，本実施例の説明の後に〔補足説明１〕として述べる。

3.2) 分割対象配列および分割対象インデックスの決定

ブロッキングをどの配列に対して実施するか，またどのインデックスで実施するかの決定プロセスを図４に示す。

【００４７】(a) 分割対象インデックスの決定では，まずブロッキング可能な候補配列およびインデックスの候補の抽出を行う。

(b) 次に，キャッシュメモリのサイズと，ブロッキング対象データのサイズとを比較する。

【００４８】(c) 配列データのアクセス距離を計算する。

(d) 以上の結果から，配列の次元数を考慮し，分割インデックスを決定する。以上の処理(a) ～(d) を繰り返す。このときに注意すべき点は以下のとおりである。

【００４９】(1) もしループの回転数が翻訳時に確定しないならば，配列の宣言を参照して，要素すべてをアクセスするとみなして計算する。

(2) ブロッキング対象の配列データの形状が翻訳時に確定しない場合，例えば“ｂ（ｉ＊８，ｋ＊８）”や“ｂ（ｃ（ｉ，ｋ））”の場合，翻訳時に正確な分割値を求めることができない。この場合には，ブロッキングを実施しない。

【００５０】(3) 次元数の考慮は，１次元配列，２次元配列，Ｎ次元（Ｎ＞＝３）配列と分離して考える。なお，この次元数の考慮については，後述する〔補足説明２〕で詳しく説明する。

【００５１】４）ブロック長の決定

ブロッキングを行うにあたり，性能を大きく左右する要因となるのが，“ブロッキング対象の配列をいくつに分割するか”である。分割のブロック長をＳＴＲＩＤＥと呼び，最適なＳＴＲＩＤＥを求めることが，ブロッキング最適化の鍵となる。このＳＴＲＩＤＥを決定する方法を，〔補足説明３〕として後述する。

【００５２】ブロック長を決定する場合の最大の要因は，ハードウェアが持つデータキャシュの大きさである。データキャシュが大きい場合，より多量のデータをデータキャシュ上に保持することが可能（ＳＴＲＩＤＥを大きくとることが可能）であり，反対に少量のデータしかキャシュ上に保持することができないときには，ＳＴＲＩＤＥを小さくすることが必要となる。以下に，ＳＴＲＩＤＥの決定要因を示す。

【００５３】(1) ループの回転数が翻訳時に確定している場合

分割後の配列がキャシュサイズに収まり，かつループの回転数で割り切れるＳＴＲＩＤＥを選択する。ブロッキング対象ループが２つあり，相互の回転数が違う場合には，最内ループの回転数に合わせて，ＳＴＲＩＤＥを決定する。また回転数が素数の場合には，配列要素が収まる最大値をＳＴＲＩＤＥとする。

【００５４】(2) ループの回転数が翻訳時に確定していない場合

回転数が翻訳時に確定しない場合にもブロッキングを実施することができる。このとき注意すべき点は，“ＳＴＲＩＤＥの合計が回転数を超える場合”を常に考慮しなければならないことである。この場合には，以下の手順でＳＴＲＩＤＥを決定する。

【００５５】ｉ）回転数が翻訳時に確定しない場合には，配列の要素すべてをアクセスするとみなし，キャシュサイズに格納可能なＳＴＲＩＤＥを決定する。方法は回転数が既知の場合と同様である。

【００５６】ii）ＳＴＲＩＤＥが決定した後，新たにループを生成する。ループの回転数が不明であるため，ＳＴＲＩＤＥと回転数との比較のためのコードを生成する必要がある。このとき，条件付転送命令を持つハードウェアでは，ＳＴＲＩＤＥと回転数との比較を条件付転送命令に置き換える。この置き換えにより，分岐のオーバヘッドを縮小することが可能となる。

【００５７】図５に，ＳＴＲＩＤＥの和とループの回転数（Ｎ）の関係を示す。この図から明らかなように，ループの回転数が不明である場合には，比較判断の命令が必要になる。

【００５８】図６は，そのためのループの終了条件を判定する命令の例を示している。例えば図６に示す“ｄｏｉ=ii,min(ii+istride-1, n)”のように，翻訳時にループの回転数（変数ｎ）が確定しない場合には，ＳＴＲＩＤＥ（変数ｉｓｔｒｉｄｅ）のそれまでの合計値と，回転数（ｎ）のうち小さいほうを選び，それをループの終了条件とする。

5）ループ変形

分割すべき配列データ，分割すべきインデックス，およびブロック長（ＳＴＲＩＤＥ）が決定した後，分割数を指示するループをオリジナルループの外に生成する。ブロッキングを実施した場合，オリジナルループと比較し

て，ループが深くなるが，最内ループのブロッキング対象となったデータがキャシュにミスヒットしないことを考慮すると，新たに生成したループのオーバヘッドは無視できる。

〔補足説明1〕　配列データの再利用解析

配列データの再利用解析では，“最内ループで使用される配列データが，どのインデックス空間で使用されるのか”のデータを収集する。このデータをインデックス・ディペンデンス・ベクトル(Index dependence vector)と呼ぶ。

【0059】例えば，図7の（イ）に示す例で説明すると，Index dependence vectorは，以下のようになる。最内ループの配列データのインデックスj，k，iと，ループの深さとの関係は，図7の（ロ）に示すようになる。この各インデックスに対応したループの深さが，Index vectorの並びになる。このケースでは，Index vectorは{j，k，i}の順番になる。ここで，インデックスが関与するものを“1”，関与しないものを“0”とし，Index dependence vectorを定める。

【0060】その結果，Index dependence vectorは，図7の（ハ）に示すように，

c(i,j) = {1,0,1} : a(i,k) = {0,1,1} : b(k,j) = {1,1,0}

となる。

【0061】これをもとに，最外ループに対して再利用(Reuse)可能な配列データを求める。各ループのインデックスのIndex vectorを定義し，Index dependence vectorとの減算を実施する。減算した結果は，結果Vectorに格納される。

【0062】Result(k1,･･･,kn) = Idv(j1,･･･,jn) − Iv(i1,･･･,in)

（n：ループの深さ）

各インデックスに対して減算の結果を求める。Result(1) = {r1,･･･,rn}，インデックスIn(1<in<n)で再利用可能なデータは，該当するIndex Vectorの欄が(' − i')の場合である。それ以外の数値の場合(' 0')は，アクセスする配列が調査対象のインデックスを含んでいため，最外ループに対して再利用不可能となる。

【0063】図7の例で説明すると，まず，

Index Dependence Vector : c(i,j) = (1,0,1)　a(i,k) = (0,1,1)

b(k,j) = (1,1,0)

を求め，次に，Index jに対して再利用可能なデータを収集する。インデックス' j' のIndex Vectorは，Iv(j)= (1,0,0)である。

【0064】Index Dependence VectorとIndex Vectorの減算を行う。

c(i,j) : (1,0,1)− (1,0,0)= (0,0,1)

a(i,k) : (0,1,1)− (1,0,0)= (-1,1,1) ★再利用可能

b(k,j) : (1,1,0)− (1,0,0)= (0,1,0)

このとき，インデックス' j' に対して，配列a（i，k）は再利用があることを意味し，インデックス' i'，' k' に対してブロッキング可能となる。

【0065】以上のように簡単なベクトルの減算により，再利用可能な配列データと，対象インデックスを求めることができる。

〔補足説明2〕　ブロッキング対象データの次元数の考慮

【1次元配列の場合】分割対象の配列が1次元のとき，分割対象となるインデックスは1つしかない。再利用可能な配列データのインデックスが最内ループのインデックスに依存していないならば，ブロッキングは実施しない。最内ループのインデックスに依存しているときのみ分割する。

【0066】【2次元配列の場合】対象となる配列データの各要素が，最内とそのすぐ外側のループのインデックスでアクセスされるならば，両方のインデックスに対して，ブロッキングを実施する。もし最内ループしか含まないとき，最内ループのインデックスのみで分割する。両方のインデックスを含まないとき，ブロッキングは実施しない。

【0067】【3次元以上配列の場合】

－最内ループに着目してブロッキングを実施する。外側ループに着目したブロッキングは翻訳コストの面から実施しない。

【0068】－ブロッキングを行うためにインデックス交換が必要なときは，ブロッキングは実施しない。最内ループのインデックスでアクセスされる配列のみブロッキングの対象とする。

〔補足説明3〕　ＳＴＲＩＤＥ（分割長）の決定

ループの回転数が翻訳時に確定している場合，分割後の配列がキャシュサイズに収まり，かつループの回転数で割り切れるＳＴＲＩＤＥ（分割長）を選択する。

【0069】(1) 正方形に分割する場合

キャシュサイズをＣ（Ｂｙｔｅ），ブロッキング対象の配列データがＮ個存在するものとし，配列の１要素の大きさをＥ（Ｂｙｔｅ）とすると，１つの配列に分配することができるキャシュサイズは

ＳＴＲＩＤＥ＝ＳＱＲＴ((Ｃ／Ｅ)／Ｎ)

の式で求められる。なお，ＳＱＲＴは，平方根を求める関数である。

【0070】回転数は翻訳時に確定しているので，分割値に近似していて，かつループの回転数で割り切れる数値が最適なＳＴＲＩＤＥになる。

❶　ブロッキング対象ループが２つあり，相互の回転数が違う場合

最内ループの回転数をベースとして，最適なＳＴＲＩＤＥ値を決定する。外側のループの回転数の決定では，回転数を超えるかどうかの判定を生成する。

【0071】❷　回転数が素数のとき

上記の式で求めたＳＴＲＩＤＥで分割する。ＳＴＲＩＤＥ値が回転数を超えたかの判定を生成する。

【００７２】(2) 最内ループのみブロッキングを実施する場合

最内ループの最適なＳＴＲＩＤＥ値をＸとして，その他のＮ－１個のループの回転数を Ic(i) (1≦i≦N－1)，対象配列の１要素の大きさをＢとすると，一つの配列要素に分配できるキャシュサイズは，

ＳＴＲＩＤＥ＝Ｃ／Ｂ＊（Π Ic(N－1)）

の式で求めることができる。これが，キャシュに収まるＳＴＲＩＤＥ長の最大値となる。以下，正方形に分割する方式と同様にして，最適なＳＴＲＩＤＥ長を決定する。次に，本発明の具体的な適用例について，図８および図９に従って説明する。

【００７３】① 図８の（イ）に示すＦＯＲＴＲＡＮソースプログラムについて，本発明による最適化を実施するものとする。なお，実際には，ソースプログラムを中間テキストに変換したものについて最適化を行うが，ここでは，説明をわかりやすくするため，ソースプログラム形式で表記する。

【００７４】このプログラムのループを検出し，タイトなループ構造であることがわかったならば，データの依存関係の情報収集を実施した後に，最内ループの配列要素のアクセスができるだけ連続となるように，インデックス交換を実施する。

【００７５】② インデックスＩが最内ループ，インデックスＪが最外ループのインデックスとなるようにループを変形する。これにより，図８の（ロ）に示すように変形される。

【００７６】③ 個々の配列要素に対して，Index dependence vector を求める。インデックスとループの深さとの関係から，Index dependence vector は，図８の（ハ）に示すようになる。

【００７７】④ 最外ループに対して，常に同じデータアクセスの軌跡を描く配列を求める。図９の（イ）に示すように，Ｊの Index vector は（1，0，0）である。これをもとに，各配列のIndex dependence vector と，Ｊの Index vector との減算を行う。この結果，Ｊのインデックス部が“－1”の配列要素，すなわちＡ（Ｉ，Ｋ）が，インデックスＪに対して，常に同じデータのアクセスの軌跡を描くデータであることがわかる。

【００７８】⑤ 図９の（ロ）に示す処理を行う。すなわち，配列Ａ（Ｉ，Ｋ）とキャッシュサイズ（Ｃとする）を比較し，Ａ（Ｉ，Ｋ）＜Ｃのとき，本最適化は必要ないので，ループ変形は実施しない。Ａ（Ｉ，Ｋ）＞Ｃのとき，Ａ（Ｉ，Ｋ）＜Ｃとなるように，Ｉ，Ｋを分割する。

【００７９】ここで，Ａ（Ｉ，Ｋ）＞Ｃであるとする。前述した方法により，分割のサイズを計算し，それぞれＩＳＴＲＩＤＥ，ＫＳＴＲＩＤＥとする。

⑥ この分割長ＩＳＴＲＩＤＥ，ＫＳＴＲＩＤＥに従って，ループ変形を実施する。すなわち，分割数を指示するループ“DO KK=1,R,KSTRIDE”，“DO II=1,N,ISTRIDE”を，元のループの外に生成する。また，Ｋ，Ｊの回転数が不明であることから，Ｋの回転数Ｒと分割数の合計，Ｉの回転数Ｎと分割数の合計の比較によるループ終了条件を定める命令を生成する。この結果，ループは，図９の（ハ）に示すように変形される。図９の（ハ）に示す最適化結果に従って，オブジェクトプログラムを生成すれば，キャッシュメモリのミスヒットが少ない性能のよいオブジェクトプログラムが得られることになる。

【００８０】

【発明の効果】以上説明したように，本発明によれば，ブロッキングを行う前に，ループ分割，インデックス交換等のループ変形を実施することにより，ブロッキングの適用範囲を広げることが可能となる。また，ブロッキングの効果により，一番実行コストが高い最内ループでのキャシュメモリへのミスヒットを減少させ，主記憶参照アクセスのオーバヘッドを削減することができるようになる。

【図面の簡単な説明】

【図１】本発明の原理説明図である。

【図２】本発明の実施例に係るループ分割によるループ再構成の例を示す図である。

【図３】本発明の実施例に係るインデックス交換の説明図である。

【図４】本発明の実施例に係る分割対象インデックス決定プロセス説明図である。

【図５】本発明の実施例に係るＳＴＲＩＤＥの合計と回転数の比較判断説明図である。

【図６】本発明の実施例に係るループの終了条件の判定説明図である。

【図７】本発明の実施例に係るループの深さ／インデックス／配列アクセスの関係説明図である。

【図８】本発明の適用例説明図である。

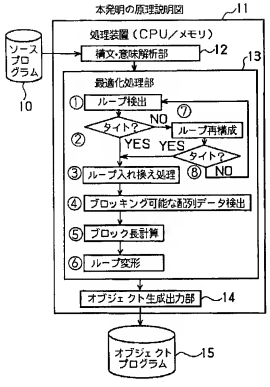
【図９】本発明の適用例説明図である。

【図１０】本発明に関係する最内ループのメモリアクセスの軌跡説明図である。

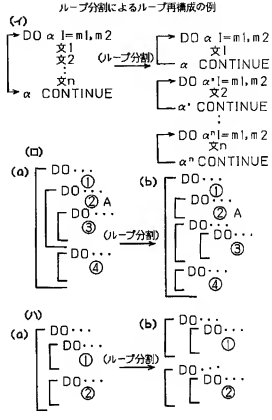
【符号の説明】

１０　ソースプログラム
１１　処理装置
１２　構文・意味解析部
１３　最適化処理部
１４　オブジェクト生成出力部
１５　オブジェクトプログラム

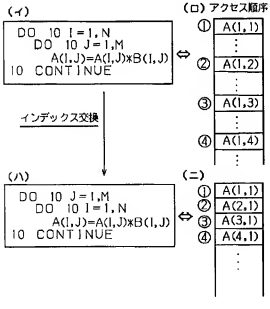
【図１】


本発明の原理説明図
11
処理装置（CPU／メモリ）
ソースプログラム
10
構文・意味解析部
12
13
最適化処理部
①ループ検出
⑦
NO
タイト？
ループ再構成
②
YES
YES
タイト？
③ループ入れ換え処理
⑧
NO
④ブロッキング可能な配列データ検出
⑤ブロック長計算
⑥ループ変形
オブジェクト生成出力部
14
オブジェクトプログラム
15

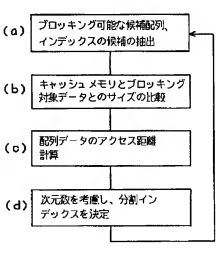
【図２】


ループ分割によるループ再構成の例
(イ)
DO α I=m1,m2
文1
文2
文n
α CONTINUE
(ループ分割)
DO α I=m1,m2
文1
α CONTINUE
DO α'I=m1,m2
文2
α' CONTINUE
DO αnI=m1,m2
文n
αn CONTINUE
(ロ)
(a)
DO…
①
DO
②A
DO…
③
DO…
④
(ループ分割)
(b)
DO…
①
DO
②A
DO…
DO…
③
DO…
④
(ハ)
(a)
DO…
DO…
①
DO…
②
(ループ分割)
(b)
DO…
DO…
①
DO…
DO…
②

【図３】

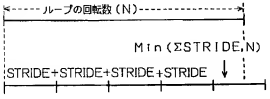

インデックス交換の説明図
(イ)
DO 10 I=1,N
DO 10 J=1,M
A(I,J)=A(I,J)*B(I,J)
10 CONTINUE
(ロ) アクセス順序
①A(1,1)
②A(1,2)
③A(1,3)
④A(1,4)
インデックス交換
(ハ)
DO 10 J=1,M
DO 10 I=1,N
A(I,J)=A(I,J)*B(I,J)
10 CONTINUE
(ニ)
①A(1,1)
②A(2,1)
③A(3,1)
④A(4,1)

【図４】


分割対象インデックスの決定プロセス説明図
(a)
ブロッキング可能な候補配列、インデックスの候補の抽出
(b)
キャッシュメモリとブロッキング対象データとのサイズの比較
(c)
配列データのアクセス距離計算
(d)
次元数を考慮し、分割インデックスを決定

【図5】

STRIDEの合計と回転数との比較判断説明図



【図6】

ループの終了条件の判定説明図

```
do ii=1, n, instride
   do j=1, m
      do k=1, r
         do i=ii,min(ii+instride-1,n)
            c(i,j)=c(i,j)+a(i,k)*b(k,j)
         enddo
      enddo
   enddo
enddo
```

【図7】

ループの深さ／インデックス／配列アクセスの関係説明図

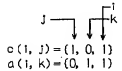
(イ)

```
do j=1, m
  do k=1, r
     do i=1, n
       c(i,j)=c(i,j)+a(i,k)*b(k,j)
enddo
```

(ロ)

Index	Depth
j	1
k	2
i	3

(ハ)



【図8】

(イ)　本発明の適用例説明図

```
DO  I=1, N
   DO  J=1, M
      DO  K=1, R
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
  CONTINUE
```

(ロ)

```
DO J=1,M
   DO K=1,R
      DO I=1,N
        C(I,J)=C(I,J)+A(I,K)*B(K,J)
  CONTINUE
```

(ハ)

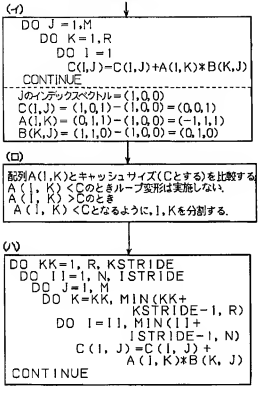
インデックス	ループの深さ
J	1
K	2
I	3

C(I,J)=(1,0,1)
A(I,K)=(0,1,1)
B(K,J)=(1,1,0)

(図9の(イ)へ)

【図9】

本発明の適用例説明図



【図10】

最内ループのメモリアクセスの軌跡説明図